J Supercomput (2006) 38:279–305
DOI 10.1007/s11227-006-8752-9

Flexible Grid service management through resource partitioning

Bruno Volckaert · Pieter Thysebaert · Marc De Leenheer · Filip De Turck · Bart Dhoedt · Piet Demeester

Abstract In this paper, a distributed and scalable Grid service management architecture is presented. The proposed architecture is capable of monitoring task submission behaviour and deriving Grid service class characteristics, for use in performing automated computational, storage and network resource-to-service partitioning. This partitioning of Grid resources amongst service classes (each service class is assigned exclusive usage of a distinct subset of the available Grid resources), along with the dynamic deployment of Grid management components dedicated and tuned to the requirements of a particular service class introduces the concept of Virtual Private Grids. We present two distinct algorithmic approaches for the resource partitioning problem, the first based on Divisible Load Theory (DLT) and the second built on Genetic Algorithms (GA). The advantages and drawbacks of each approach are discussed and their performance is evaluated on a sample Grid topology using NSGrid, an ns-2 based Grid simulator. Results show that the use of this Service Management Architecture in combination with the proposed algorithms improves computational and network resource efficiency, simplifies schedule making decisions, reduces the overall complexity of managing the Grid system, and at the same time improves Grid QoS support (with regard to job response times) by automatically assigning Grid resources to the different service classes prior to scheduling.

Keywords Grid service management · Virtual private Grid · Service Grids

1 Introduction

As more and more application types are ported to Grid environments, an evolution is noticed from purely computational and/or data Grid offerings to full-scale service Grids [1] (e.g. the EGEE Enabling Grids for E-Science in Europe project [2]). In this paper, a 'service Grid' denotes a Grid infrastructure capable of supporting a multitude of *application types* with

B. Volckaert (✉) · P. Thysebaert · M. De Leenheer · F. De Turck · B. Dhoedt · P. Demeester
Department of Information Technology, Ghent University, IMEC, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
e-mail: Bruno.Volckaert@intec.UGent.be

varying QoS levels (i.e. our definition of Service Grid is not limited to web-service enabled Grids). We use the term 'service class' as a classifier for user-submitted Grid jobs that exhibit similar resource requirements (processing requirements, I/O data requirements, priority, etc.). The architectural standards for Service Grids are provided by the Global Grid Forum's Open Grid Service Architecture (OGSA) [3], and (to a lesser extent) the Web Service Resource Framework [4], building on concepts of both Grid and Web Service communities.

Widespread Grid adoption also increases the need for automated distributed management of Grids, as the number of resources offered on these Grids rises dramatically (hence the scalability of these Grids becomes very important). Automated self-configuration and self-optimization of Grid resource usage can greatly reduce the cost of managing a large-scale Grid system, and at the same time achieve better resource efficiency, scalability and QoS support [5, 6].

The distributed service management architecture proposed in this paper can be described as a distinct implementation of the OGSA 'Service Level Manager' concept. Service Level Managers are, according to the OGSA specification, responsible for setting and adjusting policies, and changing the behavior of managed resources in response to observed conditions.

Our main goal is to automatically and intelligently assign Grid resources (both network, computing and data/storage resources) to a particular service class for exclusive use during a specified time frame (i.e. partitioning the pool of Grid resources into distinct service class-assigned resource pool subsets). The decision to assign a resource to one particular service will be based on the resources available to the Grid and monitored service class resource usage characteristics and requirements. Once resource partitioning has been performed, dedicated management components (i.e. scheduler, information service, etc.) will be associated to a service class's assigned resources, effectively constructing multiple self-managing 'Virtual Private Grids'. These Virtual Private Grids in turn improve Grid management scalability, as their management components only need to take into account the state of their partition-assigned resources along with the state and requirements of jobs from the service class they are responsible for.

In order to compare the performance of a service managed Grid with a non-service managed Grid we use NSGrid (for a detailed discussion see [7]), an ns-2 based Grid simulator capable of accurately modeling different Grid resources, management components and network interconnections. More specifically, we evaluated Grid performance (in terms of average job response time and resource usage efficiency) when different partitioning strategies are employed, and this both in case network aware as when network unaware scheduling is used.

This paper is structured as follows: Section 2 summarizes related work in this area, while Section 3 provides details on the proposed service management architecture and its interaction with other Grid components. The employed network and non network aware scheduling algorithms are highlighted in Section 4. Section 5 elaborates on the different resource partitioning strategies, while the evaluation of those partitioning strategies in a typical Grid topology is compared to a non-resource partitioned situation for varying job loads in Section 6. Finally, Section 7 presents some concluding remarks.

2 Related work

Considerable work has already been done in the area of distributed scheduling for Grids [8]. Grid scheduling taking into account service specific requirements has been dubbed *application-level scheduling*. Most notable application-level research projects include AppLeS [9] and GrADS [10].

In AppLeS, service-class scheduling agents interoperable with existing resource management systems have been implemented. Essentially, one separate scheduler needs to be constructed per application type. Our service management architecture differs from this approach in that it operates completely separated from the Grid scheduling components, working in on service-exclusivity properties located at the Information Services (responsible for storing resource properties and answering resource queries from e.g. the different schedulers).

GrADS on the other hand is a project to provide an end-to-end Grid application preparation and execution environment. Application run-time specific resource information comes from the Network Weather Service [11] and MDS2 [12]. For each application; a performance (i.e. computational, memory and communication) model needs to be provided by the user. This differs from our Service Monitor approach, which actively monitors application behavior and deduces service characteristics at run-time (see Section 3.3).

The General purpose Architecture for Reservation and Allocation (GARA) project [13] provides Globus with end-to-end Quality of Service guarantees for applications. Both advance and immediate resource reservations are supported. GARA does not offer dynamic automated resource-to-service partitioning but can instead be seen as a technology enabling the work proposed in this paper.

IBM's Tivoli Intelligent Orchestrator (TIO) and Provisioning Manager (TPM) [14] can improve service response times by monitoring registered resources and requirements for anticipated peak workloads and, if necessary, can automatically re-allocate resources in accordance with business priorities. TIO and TPM are focused on automated data center resource-to-service allocations, and require users to predefine 'optimal resource utilization' plans for each supported service class. Our service management architecture focuses on the needs of generic computational/data/service Grids, and tries to automatically (i.e. without user interaction) deduce optimal resource utilization from monitored Grid job submission behaviour.

Optimally assigning resources to services has been the subject of research in [15]. In this study however, resource selection occurs each time a job is submitted to a Grid Portal (i.e. service aware scheduling). This differs from the work proposed in this paper in which resources are pre-assigned to service classes based on service class characteristics (i.e. prior to the job scheduling process).

In contrast to the above mentioned research projects, our contribution focuses on distributed, automated and intelligent resource-to-service partitioning in a Grid environment (based on monitored service class characteristics/requirements) along with the dynamic deployment of service class exclusive management components (effectively constructing multiple Virtual Private Grids).

3 Service management concept

In this section we begin by describing the NSGrid models that are employed: Grid Site (resources, management components, etc.) and job models are discussed, along with basic job submission/resource assignment protocols. We continue by discussing the overall concept of resource-to-service partitioning in Section 3.2 and explain in Section 3.3 how our resource-to-service partitioning architecture was implemented in NSGrid.

3.1 Grid/job model

We regard a Grid as a collection of *Grid Sites* interconnected by WAN links (see Fig. 1). Each Grid Site has its own resources (computational, storage and data resources) and a

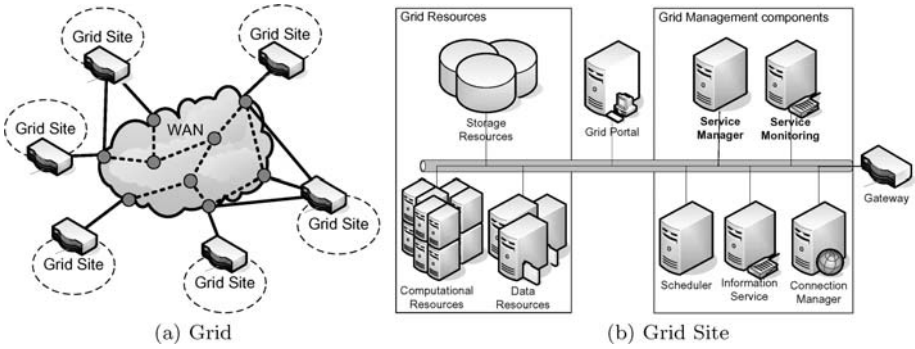


Fig. 1 Grid model

set of management components, all of which are interconnected by means of LAN links. Management components include a *Connection Manager* (capable of offering network QoS by providing bandwidth reservation support, and responsible for monitoring available link bandwidth and delay), an *Information Service* (storing registered resources' properties and monitoring their status) and a *Scheduler*. Every Grid resource in our model is given an associated service class ID property (stored in the Information Service with which the resource is registered). If no Service Management components are instantiated in the Grid, all resources' service class ID equals '0', meaning these resources can be used by *any* job (i.e. belonging to *any* service class).

The basic unit of work in our model is a *job*, which can roughly be characterized by its length (time it takes to execute on a reference processor), computational requirements (memory, operating system, installed applications, etc.), the needed input data, the output data size, the *burstiness* with which these data streams are read or written, and the service class to which it belongs. A job's service class ID can either be assigned by the Grid application from which this job was spawned (with a unique service class ID per Grid application), or alternatively jobs from different applications but with similar monitored resource requirements can be given the same service class ID by the service monitor (the latter approach is useful if one or more Grid applications spawn jobs with widely differing requirements/characteristics rendering application-based service class ID assignments less interesting). Knowing the job's total length and the frequency at which each input (output) stream is read (written), the total execution length of a job can be seen as a concatenation of instruction "blocks". The block of input data to be processed in such an instruction block is to be present before the start of the instruction block; that data is therefore transferred from the input source at the start of the previous instruction block. Similarly, the output data produced by each instruction block is sent out at the beginning of the next instruction block. We assume these input and output transfers occur in parallel with the execution of an instruction block. Only when input data is not available at the beginning of an instruction block or previous output data has not been completely transferred yet, a job is suspended until the blocking operation completes. A typical job execution cycle (one input stream and one output stream) is shown in Fig. 2. The presented model allows us to mimic both *streaming* data (high read or write frequency) and *data staging* approaches (number of input/output blocks set to 1 as can be seen in 3).

In NSGrid [7], when a simulated client submits jobs, the exact job properties are generated from pre-configured job distributions. Each Grid Site has one or more *Grid Portals* through which clients can submit their jobs. Once submitted, a job gets queued at the local *Scheduler*,

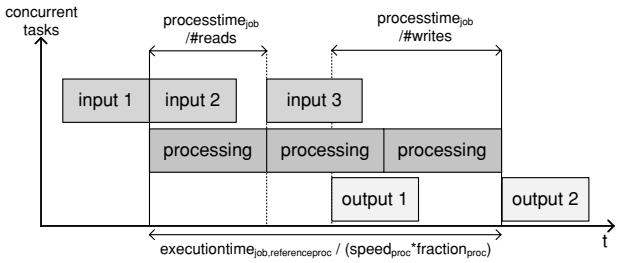


Fig. 2 Non-blocking job, simultaneous transfer and execution

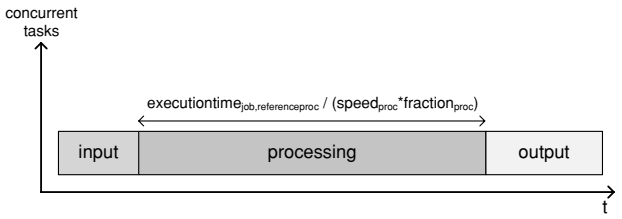


Fig. 3 Non-blocking job, pre-staged input data

which in turn queries the Information Services (IS) (located both at the local site and at foreign sites) for resources adhering to the job's requirements. Once the next scheduling round starts, the Scheduler applies one of its scheduling algorithms and (if possible) selects one or more data resources (DR) (for job input data), together with one or more storage resources (SR) (for storing job output data) and a computational resource (CR) (providing job processing power), all *not necessarily* located at one Grid Site (note that DRs and SRs can reside on the same network node—modeling one data/storage capable resource).

If the scheduling algorithm is network aware (see Fig. 8), the Connection Manager (CM) is queried for information about available bandwidth on (shortest route) paths between resources and, once a scheduling decision is made (taking into account the speed at which I/O data can be fetched/stored to/from the processing job and adjusting computational power that gets reserved for this job to match), attempts to make connection reservations between the selected resources; connection reservations provide a guaranteed minimum bandwidth available for that job. Note that reservations are not physically set up by the Connection Manager: if the bandwidth requirements of the requested connection reservation are not infringing previously guaranteed connection reservations' minimum bandwidth, the request is granted. If however this is not the case (due to the use of stale resource state information when assigning resources to jobs in the scheduling round), the connection reservation request is rejected and the job will be put back in the scheduler queue until the next scheduling round. The Connection Manager thus operates by bookkeeping all granted connection reservations and denying new reservations that would infringe on those previously granted reservations. Once all resource reservations are successful, the job is sent to the selected computational resource which takes care of fetching the different input datasets and storing the job's output data.

3.2 Resource partitioning

Our goal is to intelligently and automatically assign service class IDs to each resource so they can be used exclusively for jobs spawned from that service class. This classification of Grid resources in a per-service resource pool with its own dedicated scheduler and information service has multiple benefits:

- resource efficiency and average job response times improve (as will be shown in Section 6)
- allows for faster scheduling decisions and resource information lookups
- service class priorities can be given by assigning more resources to high-priority service classes
- locally offered service classes can be prioritized over foreign Grid Site service classes,
- reduced infrastructure costs: by allocating job loads to resources more efficiently, the number of resources can be reduced,
- improved scalability with dynamic deployment of dedicated VPG management components,
- service class dedicated management components can be finetuned to the needs of their particular service.

As we will see in Section 6, resource efficiency (and average job response times) can be improved by limiting resource availability to service classes that can make efficient use of that particular resource (e.g. taking into account service class' data locality). In addition, the number of job resource query results returned by the Information Services to the scheduler will be less than when there is one common resource pool, allowing for faster scheduling decisions (as we are in fact utilizing the resources' service class ID assignation as an advance reservation mechanism).

Of course, one has to be very careful when automatically assigning resources to service classes, as it creates the risk that certain service classes are (involuntarily) left starving for resources on which to run, while other resources are assigned to a service class for which there are no job submissions at that time (and are thus unnecessarily left idle). One also has to take into account service class necessities when making resource partitioning decisions, in order to avoid excluding a service class from access to a critical resource (e.g. prohibiting a service class access to mandatory data resources).

The same way computational, storage and data resources can be partitioned amongst different service class resource pools, network resources can also be split up by performing per-service bandwidth reservations (e.g. VPN technology). This can prevent data-intensive service classes from monopolizing network bandwidth usage and thereby hampering the performance of jobs from other service classes (see Fig. 4). Instead, each service class should automatically receive a certain bandwidth and be able to use this bandwidth without having to worry about the network usage of other services' jobs.

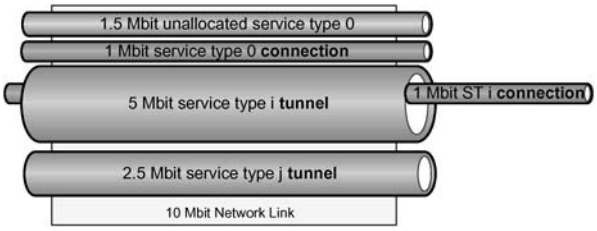


Fig. 4 Network resource partitioning

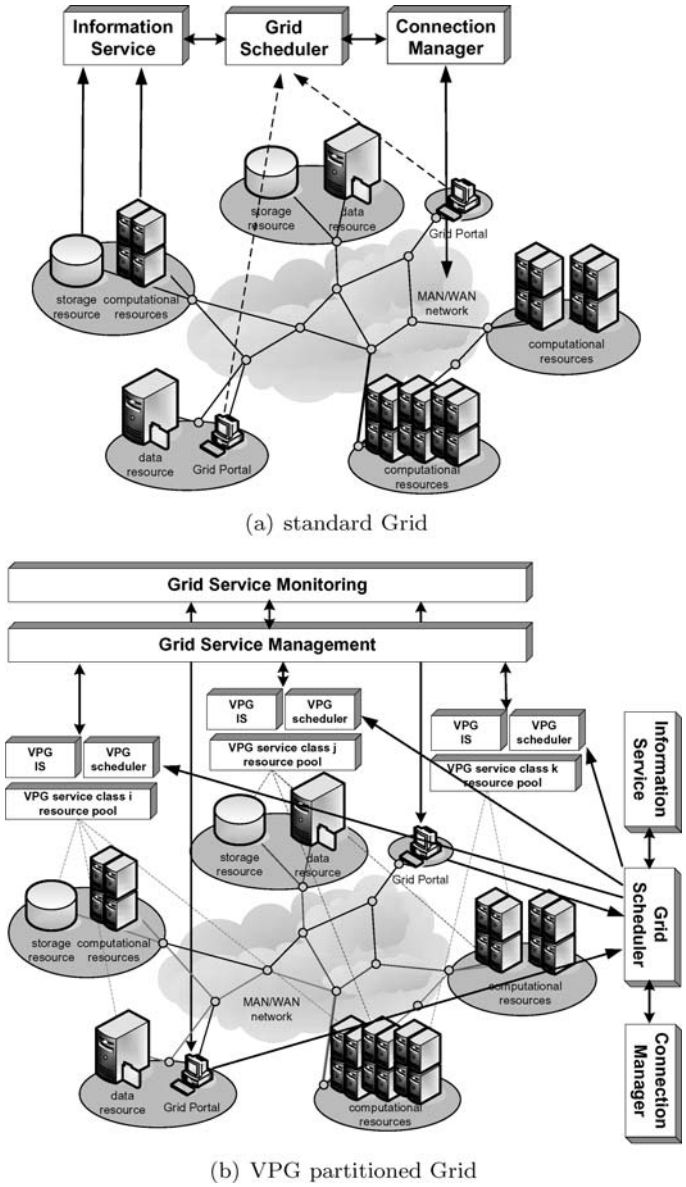


Fig. 5 Standard Grid architecture vs. Virtual private Grid partitioned Grid architecture

With combined network and resource partitioning, a Grid can be modeled as a dynamic collection of overlay Grids or *Virtual Private Grids* (VPG), with one VPG for each service class offered in the Grid. These VPGs (see Fig. 5) are not static structures in that they do not have resources assigned to them in a permanent way, but react to monitored changes in service characteristics (e.g. additional service offerings can lead to the construction of new VPGs and reallocation of resources across existing VPGs). Resource reallocation can stem from important changes in monitored service class characteristics (e.g. higher job submission

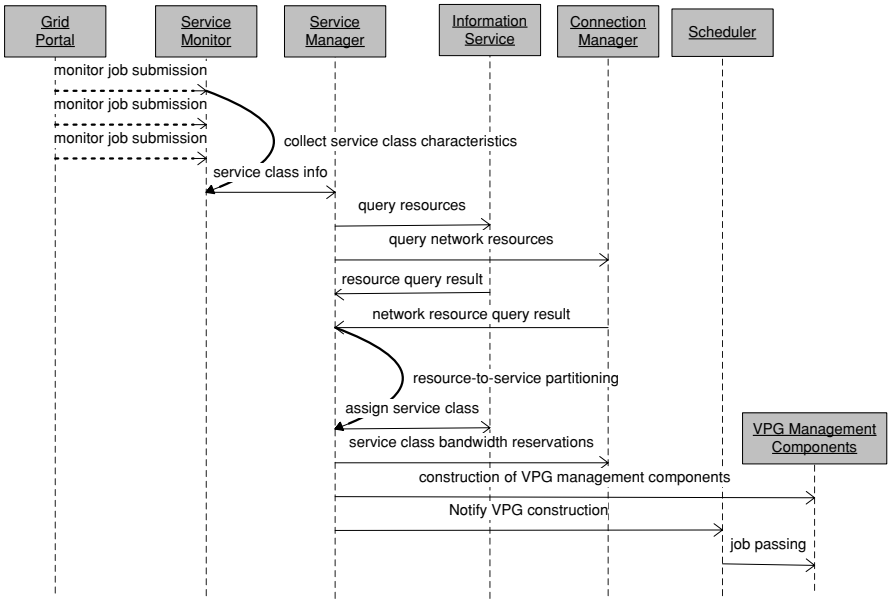


Fig. 6 VPG partitioning messages

rates for a service class), a change in service class priorities or, as already mentioned, the addition of new service classes.

3.3 NSGrid implementation

In NSGrid, a distributed service management architecture was implemented in order to evaluate the effectiveness of different resource-to-service partitioning strategies and Virtual Private Grid deployments. Each Grid Site typically has a local *Service Manager*, which interacts with the local Information Service (IS), Connection Manager (CM) and *Service Monitor* (see Fig. 7 for a sample Service Management setup in NSGrid). All NSGrid resources and management components are located at ns-2 nodes, which can be interconnected by means of different types of network links with configurable bandwidth and delay. This way, all job I/O data that is sent between the different resources is accurately simulated by ns-2, allowing us to monitor bandwidth usage, network congestion, etc.. All control messages are XML-encoded and sent over the underlying ns-2 network.

3.3.1 Service Monitor

The Service Monitor inspects job submission behavior at the Grid portals (recall that a Grid portal acts as a job submission gateway for Grid users): each time a job is submitted, job requirements (service class, priority, needed input data sets and sizes, output storage sizes, computational requirements, etc.) are extracted and overall service class properties (e.g. average job interarrival time, average I/O data sizes, average job computational needs, needed input datasets) are adjusted. When the Service Monitor has gathered adequate service class characteristics (either when service class properties remain relatively stable over a fixed period of time, or when an information dissemination timer has run out), the Service Monitor

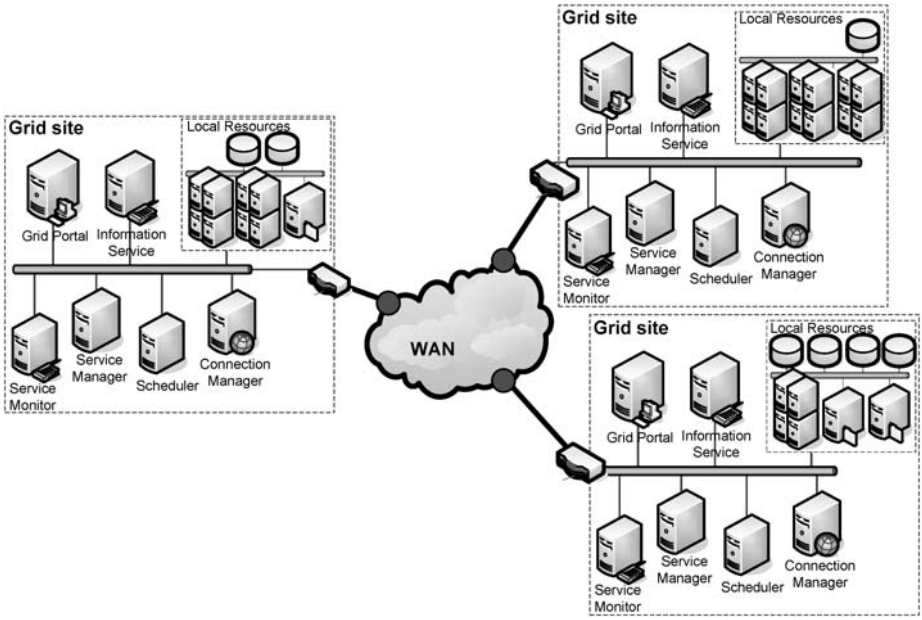


Fig. 7 NSGrid service management architecture scenario

sends the collected service class' characteristics to its known (local and foreign Grid site) Service Managers, so as to allow them to have up-to-date service class information for use by the resource-to-service partitioning algorithms. The Service Monitor keeps a record of the info that was submitted to the Service Managers, and, if substantial changes (w.r.t. a configurable threshold) in service class properties are monitored (e.g. detection of new service classes, increased service class job interarrival times, change in priority, higher job response times, etc.), sends up-to-date service class information to the Service Managers (see Fig. 6).

Note that when a job from a newly monitored service class is detected *and* all resources have been assigned to existing service classes (i.e. non service class 0 assignations), the job will have to wait for a repartitioning of resources before being able to be scheduled. The Service Monitor will wait until the newly monitored average service class characteristics have stabilized, or until its information dissemination timer has run out, after which it will contact the Service Manager informing it of the existence and characteristics of the newly monitored service class.

Each Service Monitor has a moving time window (of configurable length), such that the properties of a job that was submitted at a time before the time window's beginning are no longer taken into account when calculating service class' characteristics. In doing so, service classes that spawn no jobs during a period of time equal to the time window's length are discarded: the Service Monitor will inform the Service Manager of this occurrence, which in turn will free resources allocated to that particular service class and (if necessary) repartition.

3.3.2 Service manager

The Service Manager thus periodically receives information regarding local and foreign Grid site service class characteristics from the different Service Monitors. When the received

information does not differ (with regard to a certain threshold) from the one used to partition the Grid resources in a previous partitioning run, no resource-to-service repartitioning will occur. If however the difference between the previous values and currently monitored service characteristics (average job IAT, processing length, I/O bandwidth necessities, etc.) is too large, or if no resource partitioning has yet been done, the Service Manager will query the Information Services for the characteristics of the resources in their local Grid site resource pool. Once the answer to this query has been received, one of the resource partitioning algorithms (detailed in Section 5) is applied to the resource set, and the resulting resource partitioning solution is sent back to the Information Services, who in turn change the service class property of their registered resources. If the partitioning algorithm also works in on network resources, the Connection Manager will be contacted to make service bandwidth reservations (based on assigned computational resources, necessary input datasets and monitored service class' bandwidth requirements).

Once the partitioning algorithm has finished, resources will be assigned to service class resource pools, and (if this was not already done) dedicated Virtual Private Grid management components will be dynamically constructed and associated with the different Virtual Private Grids (in NSGrid these VPG management components are deployed at the Grid site where jobs from the VPG's service class are most common). A VPG Information Service will gather resource property and status information from all resources assigned to the VPG. This Information Service will in turn be queried by a dedicated VPG scheduler when the latter seeks information on resources adhering to a job's requirements. Note that the global (central or distributed) Grid scheduling system continues to receive all jobs submitted to the different Grid portals, but, upon inspection of the service class of each arriving job, either tries to schedule the job itself, or, when a VPG is constructed for the job's service class, immediately sends it to the dedicated VPG scheduler.

3.3.3 Information Service

Much in the same way as the Service Monitors can trigger a repartitioning of resources to services when substantial changes in service class characteristics are monitored, the Information Services are responsible for signaling changes in resource availability. Every time an existing Grid resource becomes unavailable (either because of failure or by policy), or conversely, when new resources become available to the Grid, the Information Services report this change to the Service Manager. The latter then decides if a resource-to-service repartitioning is necessary.

It is important to note that, while resources are assigned for exclusive use by a particular service, not one job using a service class reassigned resource will be interrupted (preventing jobs from being pre-empted when the CR it is running on is assigned to a different service class). The service assignment will thus only be effective for *new* jobs or jobs currently in the scheduler queue. At the time of scheduling, queries will be sent to the Information Services for resources adhering to the job's requirements, and these Information Services will return only those resources that are assigned to that particular job's service class.

4 Scheduling strategies

When jobs are submitted, a Scheduler needs to decide where to place the job for execution. The scheduling algorithm used in making this selection has a big impact on Grid performance, and influences overall Grid job throughput, Grid resource efficiency etc. All presented algorithms

are *queueing* algorithms [17], that is, whenever an algorithm is invoked, it will attempt to schedule the not-yet scheduled jobs in the order of arrival on the time-shared resources. Jobs that cannot be scheduled will be requeued, preserving the relative order of arrival (note that other requeueing methods are available). The time between two scheduling rounds can be fixed, but it is also possible to set a threshold (e.g. time limit or number of unscheduled jobs in the queue) which triggers the next scheduling round. As the goal of each algorithm is the minimization of each job's response time, a natural metric to benchmark the different algorithms is the average job *turnaround time*. In what follows we will briefly explain the different scheduling strategies used in our simulations (for a more detailed discussion see [16]). Once scheduled, our scheduler does not attempt to pre-empt jobs.

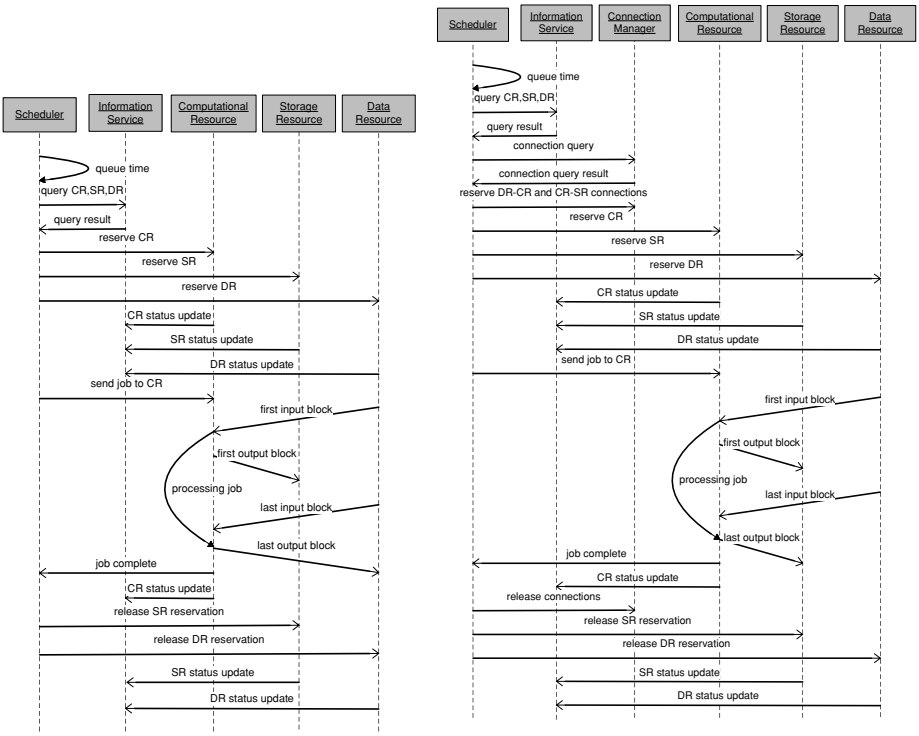
4.1 Non-network aware scheduling

Non-Network aware scheduling will compute Grid job schedules based on the status of the computational, storage and data resources (as provided by the Information Services). Algorithms that use this kind of approach will not take into account information concerning the status of resource interconnections. The decision of which resources to use for a job will be based on the information acquired from the different Information Services (i.e. job execution speed and end time will be calculated based on the status of CR/SR/DR). It is precisely because Non-Network aware algorithms assume that residual bandwidth on network links is "sufficient", that jobs can block on I/O operations: their computational progress is no longer only determined by the computational resource's processor fraction that has been allocated to it (which, together with the job's length and the computational resource's relative speed determines its earliest end time *if all input and output transfers complete on time i.e. before the start of the appropriate instruction block*), but also by the limited bandwidth available to its input and output streams. Note that the fact that network information is discarded during the scheduling implies that no connection reservations (providing guaranteed available bandwidths) are made with the connection manager—these would allow to accurately predict the job's running time.

4.2 Network aware scheduling

Network aware scheduling algorithms will not only contact the Information Services (for information about resources that adhere to the job's requirements), but will also query the Connection Manager for information about the status of the network links interconnecting these resources (i.e. the Connection Manager will send the Grid Scheduler information about connections that can be set up between DR/CR couples (necessary for job input retrieval) and CR/SR couples (needed for job output storing)). Based on the answers from the Information Services and Connection Manager, the scheduling algorithm is able to calculate job execution speed and end time more accurately, taking into account the speed at which input/output can be delivered to each available computational resource. For jobs with 1 input stream and 1 output stream, the best resource (CR/SR/DR) triplet is the one that minimizes the expected completion time of the job. This value is determined by the available processing power to that job on the computational resource (and its relative speed), the job's length, the job's total input and output data size and the residual bandwidth on the observed links from DR to CR and from CR to SR.

As explained, for some (CR,SR,DR) triplet, due to bandwidth constraints, this duration may be significantly higher than the value calculated from the job's length and the CR's relative speed, even if job execution and data transfer occur simultaneously. The scheduler



(a) Non-network aware job scheduling (b) Network aware job scheduling

Fig. 8 NSGrid non-network aware versus network aware scheduling

selects the optimal CR/DR/SR triplet and contacts the central Connection Manager to perform the necessary connection setups (the necessary bandwidth of these connections is calculated by the scheduler). The job then gets transferred to the selected CR for processing and input/output is sent from/to the DR/SR over the reserved connections. If no (local or remote) resources satisfying the job's requirements can be found, or if no connections with sufficient bandwidth are available, the job will be queued and prepared for reschedulement.
The time it takes for a job to complete since it has been submitted by the client can be broken up into:

- sending the job to the scheduler
- time spent in the scheduler's queue
- time needed for the co-allocation of resources (including network resources) allocated to that job
- transfer time for the first input data block(s)
- time needed to process the job at its allocated execution speed
- transfer time for the last output data block(s).

Each of these can be found in Fig. 8. Note that no job can become blocked because reservations are made with network resources, excluding the network from becoming an unexpected bottleneck (if the resource state information returned by the Information Services/Connection Managers and employed by the scheduler was accurate and up-to-date).

5 Partitioning strategies

The problem at hand is trying to partition resources into service class resource pools. A solution in this case is a mapping from resource to a particular service class ID, and this for all resources returned from the Service Manager—Information Service queries. A resource can also be assigned service class ID '0', meaning it can be used by jobs from every service class. Exhaustively searching for an optimal partitioning (by evaluating the fitness of a solution by means of a cost function) quickly becomes infeasible, as the amount of solutions that needs to be evaluated is $(\#serviceclasses + 1)^{\#resources}$. In our attempts to find a suitable solution in reasonable time, we have used two distinct approaches: one uses Divisible Load Theory (DLT) to obtain a tractable Integer Linear Program (ILP) modeling the service class assignment problem, while the other uses a Genetic Algorithm to obtain a resource-to-service mapping.

5.1 DLT-based partitioning

Whenever a Grid reaches a steady state (e.g. a Grid processing a periodic load), stochastic parameters regarding the distributions of job interarrival time, duration and I/O-needs can be estimated for each service class by the Service Monitoring Architecture. These parameters can then be used to populate an Integer Linear Program designed to (by assigning appropriate values for the program's decision variables)

1. Assign an exclusive service class ID to each computational resource
2. Determine the optimal *schedule* of the periodic workload over the Grid's resources, taking into account the resource-to-service assignation

An approximation used to limit the number of integer variables in this approach is to treat the aggregate workload as arbitrarily divisible (hence the name "Divisible Load Theory") [18, 19]. In this context, values of interest are $arrivals_s^n$—the computational load per time unit arriving at site s and belonging to service class n, $Sets_n$ and $Size_n$—the datasets available to service class n jobs and their respective sizes. The main decision variables in the problem are $x_{c,n}$ (binary, assigning service class n exclusive access to CR c) and $\alpha_{i,n}^c$ (real-valued, amount of service class n computational load per time unit processed at CR c which arrived at site i). Auxiliary variables needed to fulfill routing constraints on the input datasets and generated output data have been dubbed $in_{n,j}^l$ (bandwidth needed on link l for transport of dataset j of service class n) and out_s^l (bandwidth needed on link l for transport of output data to storage resource s)—note that the concept of source-based routing [20] was used to formulate the routing constraints.

Using the Divisible Load approach, the resource-to-service assignation can now be modeled as a cost minimization problem with several classes of constraints.[1]

The capacity constraints to be observed for each computational Resource and Network Link, respectively, are

$$\forall c \in CR. \sum_{i \in Sites} \sum_{n \in SC} \alpha_{i,n}^c \leq Cap_c \tag{1}$$

$$\forall l \in L. \sum_{n \in SC} \sum_{j \in Sets_n} in_{n,j}^l + \sum_{s \in SR} out_s^l \leq Cap_l \tag{2}$$

[1]Abbreviations used: GW = Gateways, L^+ = outgoing links, L^- = incoming links, Cap_c = computational res. capacity, Cap_l = link capacity.

These constraints ensure that work allocated to a Computational Resource does not exceed that resource's processing capacity, and that total network traffic over each link does not exceed that link's capacity. Network traffic is routed according to following constraints:

$$\forall n \in SC,\ j \in Sets_n.\ \sum_{d \in DR:\, j \in Sets_d} \sum_{l \in L_d^+} in_{n,j}^l = \frac{\sum_{s \in Sites} arrivals_s^n \times Size_n}{\#Sets_n} \tag{3}$$

$$\forall c \in CR,\ n \in SC,\ j \in Sets_n.\ \sum_{l \in L_c^-} in_{n,j}^l = \frac{\sum_{i \in Sites} \alpha_{i,n}^c \times Size_n}{\#Sets_n} \tag{4}$$

$$\forall c \in CR,\ s \in SR.\ \sum_{l \in L_c^+} out_s^l = \sum_{n \in SC} \alpha_{Site_s,n}^c \times Size_n \tag{5}$$

$$\forall s \in SR.\ \sum_{l \in L_s^-} out_s^l = \sum_{n \in SC} arrivals_{Site_s}^n \times Size_n \tag{6}$$

$$\forall g \in GW,\ n \in SC,\ j \in Sets_n.\ \sum_{l \in L_g^-} in_{n,j}^l = \sum_{l \in L_g^+} in_{n,j}^l \tag{7}$$

$$\forall g \in GW,\ s \in SR.\ \sum_{l \in L_g^-} out_s^l = \sum_{l \in L_g^+} out_s^l \tag{8}$$

The first two equations in this series describe how much traffic is carried on the network links departing from the Data Resources, given that any job of a given service class has an equal probability to process any of the data sets available to that service class. That same amount of network traffic is of course to be retrieved at the Computational Resource side.

The next two equations present the analogous observation for output data generated by the jobs.

The last two equations state that network flow (both for input and output data) is conserved when crossing intermediate routers.

A feasible schedule is obtained by demanding that the total distributed workload equals the size of the arriving workload per time unit:

$$\forall i \in sites,\ n \in SC.\ \sum_{c \in CR} \alpha_{i,n}^c = arrivals_i^n \tag{9}$$

Constraints concerning the exclusive reservation of each CR:

$$\forall c \in CR.\ \sum_{n \in SC} x_{c,n} = 1 \tag{10}$$

$$\forall c \in CR,\ n \in SC.\ \sum_{i \in Sites} \alpha_{i,n}^c \leq x_{c,n} \times Cap_c \tag{11}$$

where the last equation is used to express that only those Computational Resources which have been explicitly assigned to a service class may actually perform work in that service class.

The "cost" to be minimized can take on several forms; for instance, the total amount of data traveling over network links per unit of time (in the steady-state Grid) can be described

in terms of problem variables as

$$\sum_{l \in L} \left(\sum_{n \in SC,\, j \in Sets_n} in_{n,j}^{l} + \sum_{s \in SR} out_{s}^{l} \right) \tag{12}$$

Using this cost function in the ILP results in a workload schedule and service class assignation yielding minimal aggregate network load for a given arrival process. Alternatively, one can choose to minimize the maximal unused computational resource fraction, which results in an "even" workload distribution across all computational resources according to their respective capacities. This approach can be modeled by adding the constraints

$$\forall c \in CR, n \in SC.cost \geq \frac{\left(x_{c,n} \times Cap_c - \sum_{i \in Sites} \alpha_{i,n}^{c}\right)}{Cap_c} \tag{13}$$

and minimizing the cost.

5.2 Genetic algorithm heuristic

The resource class assignment can easily be encoded into an n-tuple of service class IDs, where n equals the number of resources. These *chromosomes* can then be fed to a Genetic Algorithm (GA) which evaluates the fitness of each chromosome (i.e. possible service class assignment) w.r.t. a cost function $f(x)$ (see Algorithm 5.1. Unlike with an Integer Linear Program, this cost function needs not be "linear" in the decision variables, giving this partitioning approach more expressive power than the DLT-based partitioning.

Algorithm 5.1 starts with an initial population size of m randomly generated tuples (each tuple b consisting of n service class ID slots). While the stopcondition is not fulfilled, the GA applies a proportional selection, after which a two-point crossover and a mutation step occur. The proportional selection selects tuples based on their fitness (with fitter solutions more likely to be selected and carried over to the next generation). In the next step, a two-point crossover operation is applied (for each two consecutive tuples the crossover probability ρ_C determines if all service class IDs between the randomly selected $pos1$ and $pos2$ are switched). Finally, the mutation operation is performed for each tuple, with mutation probability ρ_M determining which of the n service class ID slots needs to be mutated to a random service class ID.

Depending on how much time is available between partitioning runs (which in turn depends on the stability of the different service characteristics), parameters of this GA can be tuned in such a way that feasible search times can be attained (i.e. search time $<<$ time between partitioning runs).

In the next sections we provide details on some implemented partitioning strategies (and accompanying cost functions): Section 5.2.1 and Section 5.2.2 describe computational resource partitioning based on the processing requirements of respectively local and global service classes. Taking into account the site locality of much needed service class' input datasets is discussed in Section 5.2.3. Finally, partitioning of network resources based on data requirements of the different service classes is discussed in Section 5.2.4. We assume that the Service Manager has received both up-to-date local and foreign Grid Site service characteristics from the Service Monitors and resource properties from the Information Services.

Algorithm 5.1: GENETIC ALGORITHM(*resources*)

$population_{initial} \leftarrow (b_{(1,0)}, ..., b(m,0)),\ t \leftarrow 0$
while *stopcondition* **false**
do
- **comment:** proportional selection
- **for** $i \leftarrow 1$ **to** m
- **do**
  - $x \leftarrow rand[0,1]$
  - $k \leftarrow 1$
  - **while** $k < m$ **and** $x < \sum_{j=1}^{k} \frac{f(b_{j,t})}{\sum_{j=1}^{m} f(b_{j,t})}$
    - **do** $k \leftarrow k + 1$
  - $b_{i,t+1} \leftarrow b_{k,t}$
- **comment:** two-point crossover
- **for** $i \leftarrow 1$ **to** $m - 1$ **step** $i + 2$
- **do**
  - **if** $rand[0,1] \leq \rho_C$
    - **then**
      - $pos1 \leftarrow rand[1,n]$
      - $pos2 \leftarrow rand[1,n]$
      - **if** $pos1 > pos2$
        - **then** $switch(pos1, pos2)$
      - **for** $k \leftarrow pos1$ **to** $pos2$
        - **do** $switch(b_{i,t+1}[k], b_{i+1,t+1}[k])$
- **comment:** mutation
- **for** $i \leftarrow 1$ **to** m
- **do**
  - **for** $k \leftarrow 1$ **to** n
    - **do**
      - **if** $rand[0,1] < \rho_M$
        - **then** $b_{i,t+1}[k] \leftarrow rand[0, \#SC]$
- $t \leftarrow t + 1$

5.2.1 Local service CR partitioning

The first (and simplest) partitioning strategy only takes into account the computational processing needs and priority of the different *local* service classes. The Service Manager queries the Information Services for all local computational resources and calculates average service class' requested processing power as the average processing time of that service class (as measured on a CR running at reference speed) divided by the average interarrival time of that SC (the higher job interarrival times, the less processing power will be needed) and multiplied with the number of sites that submit jobs from this SC.[2]:

$$\forall SC \cdot ppower_{req_{SC}} = sites_{SC} \times \frac{ptime_{ref_{SC}}}{IAT_{SC}}$$

[2] $ptime_{ref_{SC}}$ = average processing time of service class SC job on reference CR, $sites_{SC}$ = amount of Grid portals launching service class SC's jobs, IAT_{SC} = average service class SC's job interarrival time.

The relative processing power assigned to a service class (sum of processing power of computational resources assigned to that SC) can be found from[3]:

$$\forall SC \cdot ppower_{asg_{SC}} = \sum_{\forall CR \in SC} \frac{speed_{CR}}{speed_{CR_{ref}}} \times ptime_{ref_{SC}}$$

Once CR query answers have been received, the GA (as shown in Algorithm 5.1) will be started with cost function $f(x)$ described in Algorithm 5.2.

Algorithm 5.2: $f_{CRpart_{local}}$x

$result \leftarrow \frac{ppower_{asg_0}}{2}$
$maxAlloc_{over} \leftarrow 0$
$maxAlloc_{under} \leftarrow 0$
for $i \in SC_{local}$
do
- $aux \leftarrow ppower_{req_i} - ppower_{asg_i}$
- **if** $aux < 0$
  - **then**
    - **if** $-aux > maxAlloc_{over}$ **then** $maxAlloc_{over} \leftarrow -aux$
    - $aux \leftarrow ppower_{asg_i}$
  - **else**
    - **if** $\frac{aux}{ppower_{req_i}} > maxAlloc_{under}$ **then** $maxAlloc_{under} \leftarrow \frac{aux}{ppower_{req_i}}$
    - $aux \leftarrow ppower_{asg_i} - aux$
- $result \overset{+}{\leftarrow} \frac{priority_i}{(\sum_{j \in SC_{local}} priority_j)} \times aux$

$result \overset{-}{\leftarrow} maxAlloc_{over} + maxAlloc_{under}$
return $(result)$

In this cost function (which is to be maximized), the objective is to donate to each *local* service class the same amount of processing power *relative* to their requested processing power (giving a higher cost function impact factor to service classes that have a high priority). The $maxAlloc_{over}$ and $maxAlloc_{under}$ parameters assure an even spread of processing power to services (both in case insufficient processing power is available and when sufficient processing power is available), as they keep track of the maximum amount of overallocated/underallocated processing power and penalize the cost function result accordingly.

5.2.2 Global service CR partitioning

The second partitioning strategy adds support for services offered at foreign grid sites. The cost function impact factor of assigning resources to foreign service classes can be adjusted by the local Service Manager by tuning the foreign service policy $\rho_{SC_{foreign}}$. Support for foreign service classes can range from no impact at all on the cost function ($\rho_{SC_{foreign}} = 0$) to an impact equal to that of local service classes ($\rho_{SC_{foreign}} = 1$) or any value in between.

[3] $speed_{CR}$ = processing speed of CR, $speed_{CR_{ref}}$ = processing speed of reference CR.

The resulting cost function is stated in Algorithm 5.3.

Algorithm 5.3: $f_{CRpart_{global}}$x

$$
\begin{array}{l}
result \leftarrow \frac{ppower_{asg_0}}{2} \\
maxAlloc_{over} \leftarrow 0 \\
maxAlloc_{under} \leftarrow 0 \\
\textbf{for } i \in SC_{local} \cup SC_{foreign} \\
\quad \textbf{do} \left\{ \begin{array}{l}
aux \leftarrow ppower_{req_i} - ppower_{asg_i} \\
\textbf{if } aux < 0 \\
\quad \textbf{then} \left\{ \begin{array}{l} \textbf{if } -aux > maxAlloc_{over} \\ \quad \textbf{then } maxAlloc_{over} \leftarrow -aux \\ aux \leftarrow ppower_{asg_i} \end{array} \right. \\
\quad \textbf{else} \left\{ \begin{array}{l} \textbf{if } \frac{aux}{ppower_{req_i}} > maxAlloc_{under} \\ \quad \textbf{then } maxAlloc_{under} \leftarrow \frac{aux}{ppower_{req_i}} \\ aux \leftarrow ppower_{asg_i} - aux \end{array} \right. \\
\textbf{if } i \in SC_{foreign} \\
\quad \textbf{then } aux \leftarrow aux \times \rho_{SC_{foreign}} \\
result \stackrel{+}{\leftarrow} \frac{priority_i}{(\sum_{j \in SC} priority_j)} \times aux
\end{array} \right. \\
result \stackrel{-}{\leftarrow} maxAlloc_{over} + maxAlloc_{under} \\
\textbf{return } (result)
\end{array}
$$

5.2.3 Input data locality penalization

Resource partitioning based solely on the processing needs of the different services can lead to bad performance. In case of data-intensive services in particular, one wants these services to be processed on computational resources located near input data that is generally requested by those service classes. In order to provide this functionality, the Service Manager queries the Information Services for both computational and data resources and constructs a list of which CRs have local access (i.e. accessible from the local Grid Site) to which input sets. We adjust the cost function to include this notion and penalize assigning a computational resource that has *no local access to an input dataset much-needed by the assigned service*. The actual penalty depends on the input data intensiveness of the service class i ($\frac{InputReq_i}{IAT_i}$) when compared to the total input data requirements of all service classes ($\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}$):[4]

$$cost_{CR \in SCi} = \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \frac{\rho_{cost}}{\#CR_{assigned_i}}$$

An additional (yet larger) penalty is given when, amongst all computational resources assigned to a particular service, *not one of them* has access to a needed dataset, as it can be considered best practice that at least one computational resource can access a needed input

[4] $InputReq$ = avg. service class's input size requirement, $\#CR_{assigned}$ = amount of CRs assigned to service class, ρ_{cost} = data non-locality penalty factor.

set locally. This cost is only charged once for each service class.

$$cost = \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \rho_{cost}$$

Both costs can be used as a penalty for the cost function in Algorithm 5.2 and 5.3.

5.2.4 Network partitioning

Since the Service Monitor keeps track of I/O data characteristics of each service, data intensiveness relative to the other services can be calculated. This in turn can be used to perform per-service network bandwidth reservations. We have implemented a proof-of-concept network partitioning strategy, in which the Service Manager calculates average data requirement percentages for each service class i[5]

$$bw_{req_i} = \frac{\frac{bw_{input_i} + bw_{output_i}}{IAT_i}}{\sum_{\forall j \in SC} \frac{bw_{input_j} + bw_{output_j}}{IAT_j}}$$

and passes this information to the Connection Manager, who in turn will make service class bandwidth reservations on all network links for which it is responsible. Network partitioning can be applied to all previously mentioned partitioning algorithms.

6 Performance evaluation

6.1 Resource setup

A fixed Grid topology (see Fig. 9) was used for all simulations (run on an LCG-2.6.0 Grid [21] comprised of dual Opteron 242 1.6 Ghz worknodes with 2 GB RAM per cpu, and operating under Scientific Linux 3). First, a WAN topology (containing 9 core routers with an average out-degree of 3) was instantiated using the *GridG* tool [23]. Amongst the edge LANs of this topology, we have chosen 12 of them to represent a Grid site. Each site has its own resources, management components and Grid portal interconnected through 1 Gbps LAN links, with Grid site interconnections consisting of dedicated 10 Mbps WAN links. A single Service Manager was instantiated, and was given access to the different Grid Sites' Information Services.

We have assigned 3 computational resources to each Grid Site (for a total of 36 CRs). To reflect the use of different tiers in existing operational Grids, not all CRs are equivalent: the least powerful CR has two processors (which operate at the reference speed). A second class of CRs has four processors, and each processor operates at twice the reference speed. The third—and last—CR type contains 6 processors, each of which operates at three times the reference speed. Conversely, the least powerful type of CR is three times as common as the most powerful CR, and twice as common as the middle one (for a total of 18 reference CRs,

[5] bw_{input} = avg. service class's input bandwidth need: $\frac{speed_{CR}}{speed_{CR_{ref}}} \times \frac{InputReq}{ptime_{ref}}$, bw_{output} = avg. service class's output bandwidth need: $\frac{speed_{CR}}{speed_{CR_{ref}}} \times \frac{OutputReq}{ptime_{ref}}$

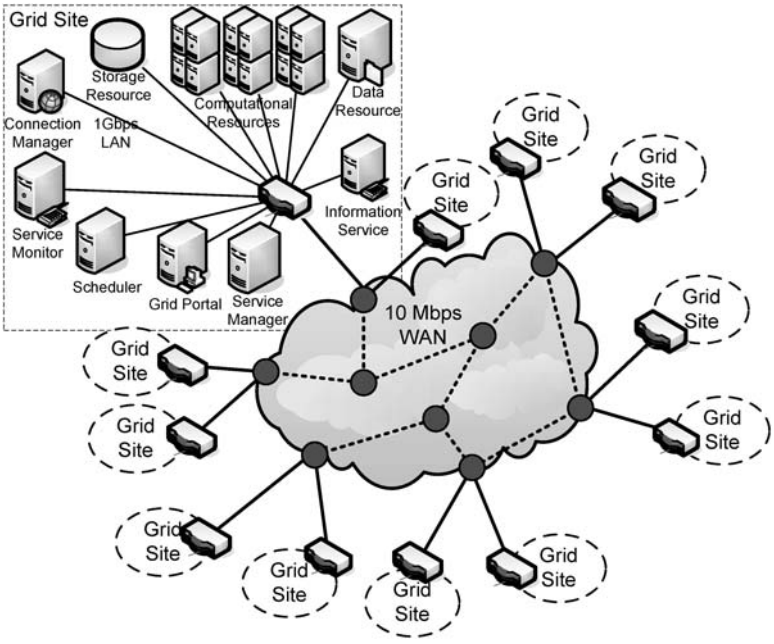


Fig. 9 Simulated multi-site Grid topology

12 four-processor CRs and 6 of the most powerful CRs deployed in our simulated topology). It is assumed that all processors can be time-shared between different jobs.

We have assumed that storage resources offer "unlimited" disk space, but are limited in terms of access/write speed by the bandwidth of the link connecting the resource to the Grid Site. Each site has at its disposal exactly one such SR. Each site's data resource contains 6 out of 12 possible data sets. These data sets are distributed in such a way that 50% of the jobs submitted to a site can have local access to their needed data set.

6.2 Job parameters

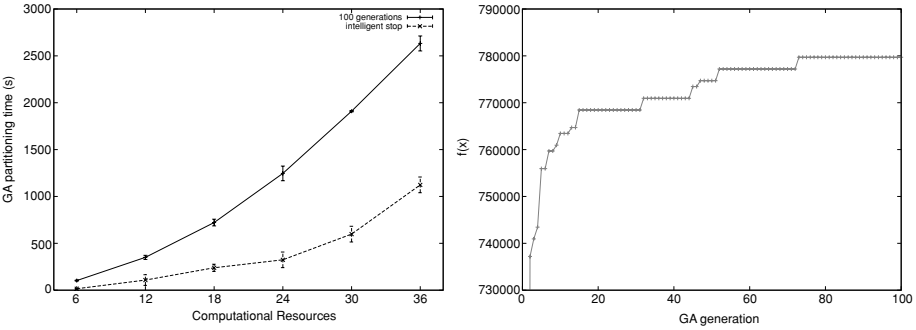
We have used two different, equal-priority service classes (each accounting for half of the total job load) in our simulations; one is more data-intensive (i.e. higher data sizes involved), while the other is more cpu-intensive. At *each* Grid Site, two "clients" have been instantiated, one for each job type. Each client submits mutually independent jobs to its Grid Portal. All jobs need a single data resource and a single storage resource. The ranges between which the relevant job parameters vary have been summarized in Table 1. In each simulation, the job load consisted of 2784 jobs. For each scheduling algorithm, we chose to use a fixed interval of 20*s* between consecutive scheduling rounds. From the arrival rates in Table 1 and the fact that multiple sites submit job simultaneously, we are likely to find multiple jobs in the queue at the start of each scheduling round.

6.3 Comparison of DLT- and GA-based partitioning

In general, our GA-based partitioning strategy provides more functionality, as it is able to support different priority schemes, shared resources (service class 0 assignations) and

Table 1 Relevant service class properties

	CPU-Job	Data-Job
Input(GB)	0.01–0.02	1–2
Output(GB)	0.01–0.02	1–2
IAT(s)	30–40	30–40
Ref. run time(s)	100–200	40–60



(a) solution calculation times (b) optimal fitness trend

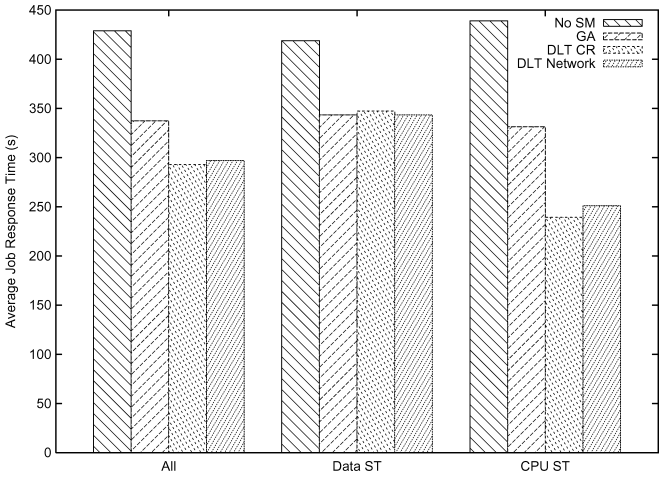
Fig. 10 Genetic algorithm measurements

local vs. foreign service differentiation. Its main drawback is the time needed to complete a GA run (with reasonable results); on our sample scenario, a naive stop condition of 100 generations takes on average 2632*s* (26.32*s* per generation but it should be noted that this time is not exclusive for GA solution calculation, but is also spent on all other simulation tasks during partitioning) as can be seen in Fig. 10(a). More reasonable GA calculation times (with an average of 1123.4*s* can however be obtained when using a more intelligent stop condition (i.e. stop when over a period of 15 generations the cost function optimum changes by less than 0.5%). The DLT-based approach on the other hand needs on average only 10*s*. For the GA approach, we used Grefenstette's settings [24], with a population of 30 per generation, $\rho_C = 0.9$ and $\rho_M = 0.01$. In case faster partitioning times need to be attained, one can either tune GA parameters (smaller population sizes, faster stopping condition, etc.) or deploy a Service Monitor/Service Manager at every Grid Site, who are then responsible for communicating with the foreign site's Service Monitor components and partitioning the resources at their assigned site (as described in Section 3.3).

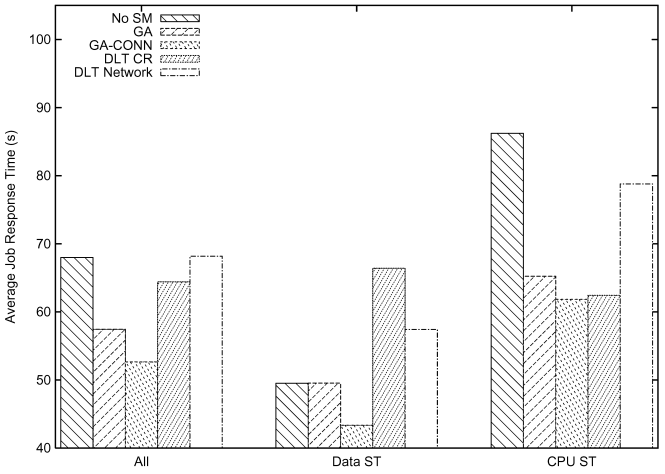
Figure 10(b) shows the trend of the cost function optimum for different GA generations (partitioning occurred on the topology discussed in Section 6.1). The cost function used is the one discussed in Section 5.2.1 (Local Service CR partitioning) with Input Data Locality penalization. It is important to note that during the calculation of a resource-to-service partitioning, Grid operation does not stall but continues as normal, as the Service Management components do not block any other management components.

6.4 Job response time

We define the *response time* of a job as the difference between its end time (time at which the job's final output block has been sent to the scheduler-assigned Storage Resource) and the time it is submitted to the scheduler. In Fig. 11 we present this average job response time for different scenarios, comparing both the (DLT & GA-based) Service Managed



(a) Non Network Aware scheduling



(b) Network Aware scheduling

Fig. 11 Job response times

versus the non-Service Managed case (DLT CR attempts to minimize Eq. (13) while DLT Network minimizes Eq. (12) as explained in Section 5.1) while at the same time evaluating the different partitioning strategies discussed in previous sections for both network aware (see Fig. 11(b)) and non network aware (see Fig. 11(a)) scheduling algorithms. The results show that average job response times can be improved significantly (by 40.44% when non network aware scheduling is used and by 22.6% when network aware scheduling is employed) by employing a resource partitioning algorithm prior to scheduling. This behavior can be explained because resources are reserved for exclusive use by a service class. It is this service-exclusivity that forces the scheduler to not assign jobs to less-optimal resources (e.g. non-local access to needed input data, low processing power available,. . .), but to keep the job in the scheduling queue until a service-assigned resource becomes available.

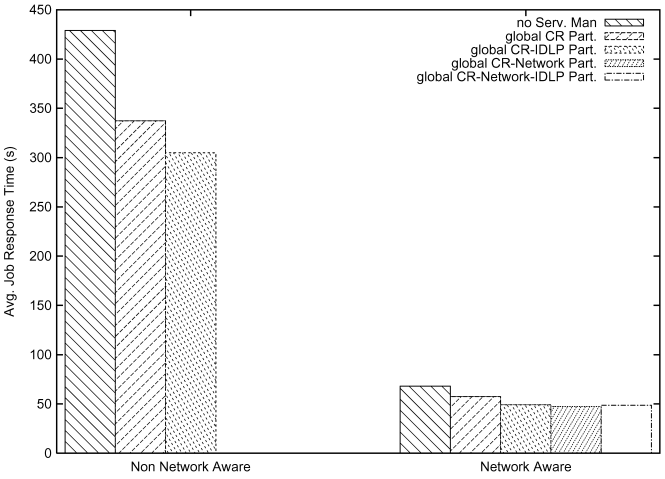


Fig. 12 Job response times for GA-based partitioning heuristics

It is noteworthy that the DLT-based partitioning works best when network unaware scheduling algorithms are used (especially for the computationally intensive service class), as it outperforms the slower GA-based partitioning strategies. However, when network aware scheduling strategies are employed (leading to much lower overall job response times as the scheduler takes into account the state of the network links interconnecting the various resources at the moment of scheduling), the GA-based methods (particularly the GA based computational and GA-CONN computational/network resource partitioning algorithms) provide the best results.

If we compare the performance of the different GA-based partitioning heuristics (see Fig. 12) (note that when non network aware scheduling is employed, no connection partitioning results are shown, due to the fact that the non network aware scheduling algorithm does not take into account the connection reservation system) we notice that average job response times always improve when resources are partitioned amongst service classes. When scheduling non network aware, the best results are attained when using computational partitioning taking into account input data locality, as data intensive jobs can be run on computational resources reserved physically near resources that store much needed I/O data, leading in turn to less computational stalling, as I/O data suffers from less network bottlenecking. When network aware scheduling is employed, one is best of using a heuristic that partitions both computational and network resources. Network partitioning assures that service classes with high I/O requirements do not consume all bandwidth (thereby preventing computationally intensive service classes from retrieving their I/O), but instead force them to only use a predefined percentage of bandwidth.

6.5 Resource efficiency

Using the same job load, the average hopcount over which data was transferred by data-intensive jobs (with hopcount equaling the amount of hops between data resource and computational resource added to the amount of hops between computational resource and storage resource) is shown in Fig. 13. We notice that average hopcount dropped by 4.8% when network unaware scheduling was employed (computational resource partitioning with data locality

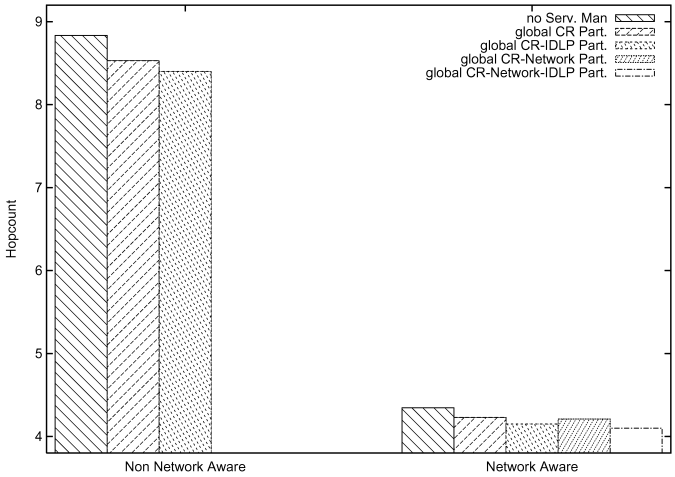


Fig. 13 Network resource efficiency

versus non-service partitioned resources), and by 5.5% when a network aware scheduling heuristic was used (network partitioning with data locality compared to the non-service managed case), due to the fact that input/output data was located at resources closer to the job's service class' assigned CRs. Network resources are thus used most sparingly when computational and network resource partitioning with input data locality is employed together with a scheduling algorithm that takes into account the state of the network links interconnecting the job's resources.

Furthermore, we calculated the average computational resource utilization:

$$\frac{\sum_{j \in Jobs_{CR}} Load_j}{Makespan \times speed_{CR}}$$

The improvement obtained by employing resource-to-service partitioning when using network unaware scheduling equals 17%, whereas in the case where network aware scheduling is used, it is 14.6%. Indeed, the fastest (and rarest in our topology) computational resources were automatically reserved for processing computationally complex jobs, disallowing data intensive jobs from cluttering these resources and using their full processing potential for those computationally intense jobs. The slower computational resources were then assigned to the data intensive service classes, who, because of their large I/O needs benefit more from having fast (i.e. LAN) access to much needed data.

6.6 Scheduling

We measured the time it takes to calculate a scheduling decision and noticed a decrease in scheduling time of 28.17% when comparing the service managed Grid to the non-service managed Grid in case network aware scheduling is used (i.e. from an average 7.88*s* in the non service managed case to 5.66s in the service managed Grid). This behaviour can be explained by the fact that a scheduler queries the Information Services for resources adhering to a job's requirements *and* assigned to either the job's service class or service class 0. When resources

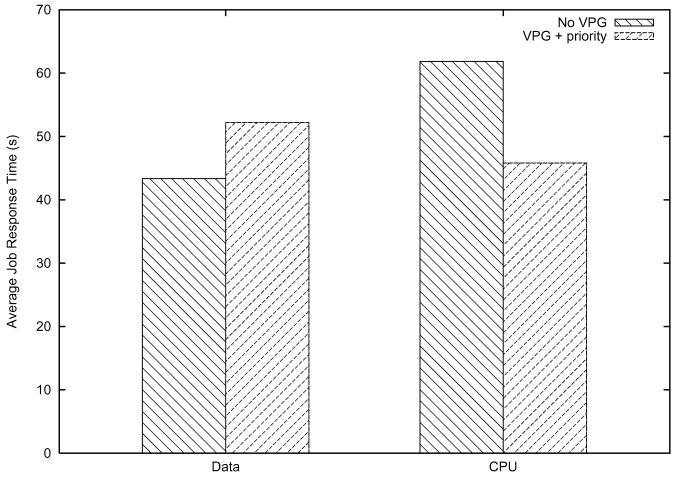


Fig. 14 VPG service class priority support

are partitioned amongst services, less results will be returned to the scheduler, allowing for faster schedule making decisions.

6.7 Priority - Service class QoS support

In another experiment, we gave the cpu-intensive jobs higher priority than the data-intensive jobs and let the Service Manager construct a Virtual Private Grid (dedicated resource pool, scheduler and information service) for each service class. Due to the high priority of the cpu-intensive class, its cost function impact factor becomes higher which leads to more (and/or better) resources being assigned to the prioritized class. Also, during deployment of the VPG schedulers, the Service Manager configures the dedicated cpu-intensive scheduler to schedule those prioritized jobs as soon as possible, using a network aware scheduling algorithm (the data intensive jobs were also scheduled using a network aware scheduling algorithm, but were by default queued until the next scheduling round). The results are shown in Fig. 14: the average job response time of the computationally intensive service class is substantially improved (due to more/better resources assigned to this service class and the ASAP scheduling policy enforced by the VPG scheduler), while the data intensive service class's average response time gets worse (prioritizing service classes over other service classes can not lead to win-win situations: the non-prioritized service classes' performance will deteriorate).

7 Conclusions

We proposed the use of a distributed service management architecture, following the OGSA 'service level manager' concept, capable of monitoring service characteristics at run-time and partitioning Grid resources amongst different priority service classes. This partitioning, together with the dynamic creation of per-service management components, lead to the introduction of the Virtual Private Grid concept. A variety of resource-to-service partitioning algorithms (some based on Divisible Load Theory and others employing Genetic

Algorithm heuristics) were discussed and we evaluated their performance on a sample topology using NSGrid. Our results show that the proposed service management architecture improves both network and computational resource efficiency and job turnaround times, eases the process of making scheduling decisions, and at the same time offers service class QoS support. Management complexity and scheduling/information service scalability is improved due to the automated deployment of service class dedicated management components.

Acknowledgments Bruno Volckaert and Marc De Leenheer would like to thank the Institute for the Promotion and Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Pieter Thysebaert and Filip De Turck are research assistant and postdoctoral fellow, respectively, funded by the Research Foundation - Flanders (FWO-Vlaanderen).

References

1. Foster I, Kesselman C, Nick JM, Tuecke S (2002) Grid services for distributed system integration. IEEE Computer 35(6):37–46
2. Enabling Grids for E-Science in Europe. website, `http://egee-intranet.web.cern.ch`
3. Foster I et al (2005) The open Grid services architecture, version 1.0. draft-ggf-OGSA-spec-019, `http://forge.gridforum.org/projects/ogsa-wg`
4. Czajkowski K et al., The WS-Resource framework version 1.0. draft, `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`
5. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Computer 36(1):41–50
6. Ganek AG, Corbi TA (2003) The dawning of the autonomic computing era. IBM Systems Journal 42(1):5–18
7. Volckaert B, Thysebaert P, De Turck F, Demeester P, Dhoedt B (2003) Evaluation of Grid scheduling strategies through a network-aware grid simulator. In: Proc. of PDPTA 2003 Vol. 1, pp. 31–35
8. Ranganathan K, Foster I (2003) Simulation studies of computation and data scheduling algorithms for data grids. Journal of Grid Computing 1(1):53–62
9. Berman F et al. (2003) Adaptive computing on the Grid using apples. IEEE Transactions on Parallel and Distributed Systems 14(4):69–382
10. Dail H, Berman F, Casanova H (2003) A decoupled scheduling aproach for grid application development environments. Journal of Parallel and Distributed Computing 63(5):505–524
11. Wolski R, Spring N, Hayes J (1999) The network weather service: A distributed resource performance forecasting service for metacomputing. Journal of Future Generation Computing Systems 15(5–6)
12. Czajkowski K, Fitzgerald S, Foster I, Kesselman C (2001) Grid information services for distributed resource sharing. In: Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing
13. Foster I, Roy A, Sander V (2000) A quality of service architecture that combines resource reservation and application adaptation. In: Proc. of the eighth international workshop on quality of service (IWQoS 2000)
14. Rodger A (2004) Analyst report: Butler group subscription services: Technology infrastructure—IBM Tivoli intelligent orchestrator and IBM tivoli provisioning manager. `ftp://ftp.software.ibm.com/software/tivoli/analystreports/ar-orch-prov-butler.pdf`
15. Lee HL, et al (2004) A resource manager for optimal resource selection and fault tolerance service in Grids. CCGrid 2004, pp 572–579
16. Thysebaert P, Volckaert B, De Turck F, Dhoedt B, Demeester P (2004) Network aspects of Grid scheduling algorithms. In: Proc. of PDCS 11
17. Hovestadt M, Kao O, Keller A, Streit A (2003) Scheduling in HPC resource management systems: Queueing vs. planning. In: Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, Feitelson DG, and Rudolph L (eds), Springer LNCS 2862
18. Yu D, Robertazzi TG (2003) Divisible load scheduling for Grid computing. In: Proc. of the IASTED 2003 International Conference on Parallel and Distributed Computing and Systems (PDCS)
19. Thysebaert P, De Turck F, Dhoedt B, Demeester P (2005) Using divisible load theory to dimension optical transport networks for computational grids. In: Proc. of OFC/NFOEC
20. Kitatsuji Y, Kobayashi K, Kitamura Y et al (2002) Deployment of APAN Tokyo XP and evaluation of source based routing. Transactions of the Institute of Electronics, Information and Communication Engineers B J85-B(8):1164–1171

21. LHC Computing Grid project, website, http://lcg.web.cern.ch/LCG
22. The Network Simulator-NS2, website, http://www.isi.edu/nsnam/ns
23. Lu D, Dinda P (2003) GridG: Generating realistic computational Grids. Performance Evaluation Review 30(4)
24. Grefenstette JJ (1986) Optimization of control parameters for genetic algorithms. IEEE Trans. Systems, Man, and Cybernetics 16(1):122–128